Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Urban Škvorc

Razvoj knjižnice za medprocesno komunikacijo v interaktivnih sistemih

diplomsko delo

univerzitetni študijski prve stopnje
računalništvo in informatika

Mentor: doc. dr. Danijel Skočaj

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V interaktivnih sistemih je zelo pomemben hiter odziv za učinkovito komunikacijo z uporabnikom. Zato morajo vsi deli interaktivnega sistema delovati učinkovito. V sistemih, kjer se vzporedno izvaja več procesov, je potrebno zagotoviti tudi učinkovito medprocesno komunikacijo. V diplomski nalogi razvijte enostaven in učinkovit sistem za medprocesno komunikacijo v operacijskem sistemu Linux. Implementirana knjižnica naj bo lahka in enostavna za uporabo, zagotavlja pa naj hitro izmenjavo podatkov med procesi. Razvito knjižnico tudi ovrednotite in primerjajte s sorodnimi rešitvami.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Urban Škvorc sem avtor diplomskega dela z naslovom:

Razvoj knjižnice za medprocesno komunikacijo v interaktivnih sistemih

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Danijela Skočaja
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 14. septembra 2015 Podpis avtorja:

Zahvaljujem se svoji družini, mentorju in Luki Čehovinu, ki je pomagal pri praktični izgradnji programa.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
RPC	Remote Procedure Call	klic oddaljenega podprograma
IPC	Inter Process Communication	medprocesna komunikacija
POSIX	Portable Operating System Interface	prenosni vmesnik za operacijske sisteme
LCM	Lightweight Communications and Marshalling	sistem za nizko zahtevno komunikacijo in serializacijo
UDP	User Datagram Protocol	protokol za uporabniška sporočila
SWIG	Simplified Wrapper and Interface Generator	Poenostavljen izdelovalnik ovojnic in vmesnikov
ISO	International Organization for Standardization	Mednarodna organizacija za standardizacijo
IEEE	Institute of Electrical and Electronics Engineers	Inštitut inženirjev elektrotehnike in elektronike
DARPA	Defense Advanced Research Projects Agency	Agencija za napredne obrambne analize
ROS	Robot Operating System	Operacijski sistem za robote

Povzetek

Cilj diplomske naloge je izdelava enostavnega in učinkovitega sistema za medprocesno komunikacijo v operacijskem sistemu Linux. Sistem omogoča razvijalcem programske opreme enostaven način za prenašanje podatkov med procesi. Izdelani sistem uporablja princip pošiljateljev in naročnikov, ki omogoča lažjo organizacijo komunikacije. Za prenos podatkov sistem uporablja vtičnice operacijskega sistema Linux, ki omogočajo učinkovit prenos podatkov med dvema procesoma. Sistem deluje na osnovi centralnega procesa, ki se izvaja v ozadju in skrbi za sprejemanje, naslavljanje in pošiljanje sporočil naročnikom. Izdelani sistem je bil v sklopu diplomske naloge primerjan z nekaterimi pogosto uporabljenimi sistemi, ki so trenutno na voljo razvijalcem. Primerjana je tako učinkovitost in hitrost pošiljanja kot tudi enostavnost uporabe in podprtost različnih platform ter programskih jezikov.

Ključne besede: medprocesna komunikacija, vtičnice, serializacija.

Abstract

The goal of the diploma thesis was to develop a simple and efficient interprocess communication system for the Linux operating system. The developed system gives developers a simple to use process for sharing information between multiple running processes. It uses a publish/subscribe model, which enables easier organization of messages. The developed system uses Linux sockets, which allow for efficient one-to-one data transfer. It is based on a central process which runs in the background and is responsible for receiving, addressing and sending messages to recipients. The developed system was compared to some of the currently available systems. The comparison addressed both the speed and efficiency of the data transfer, as well as the ease of use and the number of supported platforms.

Keywords: interprocess communication, sockets, serialization.

Poglavje 1

Uvod

Medprocesna komunikacija omogoča deljenje podatkov med različnimi procesi, ki so med seboj ločeni glede dostopa do pomnilnika. Medprocesna komunikacija je danes podprta v vsakem splošno namenskem operacijskem sistemu. Kljub temu so pri deljenju podatkov mnogokrat zaželene dodatne lastnosti, ki jih operacijski sistem neposredno ne nudi (serializacija, napredni načini naslavljanja podatkov in mnoge druge). Zato je danes za medprocesno komunikacijo razvijalcem na voljo veliko število različnih knjižnic, ki te dodatne lastnosti ponujajo.

Cilj diplomske naloge je izdelava enostavnega in učinkovitega sistema oziroma knjižnice za medprocesno komunikacijo. Večina splošno uporabljenih knjižnic namreč uvede veliko število dodatnih funkcionalnosti, ki je v nekaterih primerih nepotrebna in upočasni dejanski prenos podatkov, tako v pasovni širini prenosa kot v času dostave sporočila. Hiter prenos sporočil je zelo pomemben predvsem v interaktivnih sistemih, ki se na uporabnikov vnos odzivajo v realnem času, saj tu počasno pošiljanje sporočil pomeni neodzivno uporabniško izkušnjo. Vendar kljub razširjenosti knjižnic za medprocesno komunikacijo ni veliko takih, ki bi bile zasnovane prav za takšne sisteme.

Cilj razvitega sistema je čim večji poudarek na enostavnosti, tako v programski arhitekturi kot v uporabnosti s strani programerja ki sistem uporablja. Izdelani sistem naj tako ne bi potreboval nobenih dodatnih odvisnosti

(razen opcijskih razširitev z obstoječimi knjižnicami). Prav tako mora biti uporaba sistema v novih programih enostavna. Sama arhitektura rešitve naj, razen omogočanja uporabe sistema pošiljateljev in naročnikov za lažje strukturiranje prenosa podatkov, ne bi vsebovala naprednih rešitev, ki bi prenos podatkov naredila kompleksnejši in s tem daljši. Predvsem je poudarek na čim manjši režiji ob prenosu podatkov, ki jo veliko obstoječih sistemov uvede na primer z napredno serializacijo ali naprednim urejanjem sporočil pred prenosom.

Najprej so v diplomski nalogi na kratko predstavljene osnove medprocesne komunikacije. Opisani so različni načini, ki jih lahko razvijalci programske opreme uporabijo za deljenje podatkov med procesi ter njihove prednosti in slabosti. Opisan je princip serializacije podatkov, ki je ključnega pomena za dosledno pošiljanje podatkov med poljubnimi procesi. Opisu sledi kratka predstavitev nekaterih izbranih storitev za medprocesno komunikacijo, ki so v nadaljevanju primerjane z izdelanim programom. V drugem poglavju je predstavljen izdelani sistem, tako s stališča arhitekture kot s stališča uporabe. Opisani so glavni cilji, ki so bili zastavljeni pred izdelavo sistema, in programski pristopi, ki so bili za doseg teh ciljev uporabljeni. Priložen je tudi kratek programski osnutek, ki prikaže uporabo sistema v praksi. Sledi še primerjava s tremi izbranimi sistemi, ki so glede funkcionalnosti podobni izdelanemu sistemu. Predstavljena je primerjava hitrosti prenosa podatkov ter čas režije posameznega paketa. Na kratko so primerjane še težavnost uporabe in namestitve ter podprti operacijski sistemi in programski jeziki.

Poglavje 2

Medprocesna komunikacija

Pojem medprocesna komunikacija opisuje aktivnost, pri kateri več ločenih procesov med seboj deli podatke. V tem poglavju bodo na kratko opisani pogosti načini, ki na nivoju operacijskega sistema to omogočajo. Čeprav pod medprocesno komunikacijo štejemo tudi izmenjavo podatkov med procesi, ki delujejo na ločenih računalniških enotah, takšni sistemi ne bodo opisani, saj je poudarek diplomske naloge na deljenju podatkov med procesi, ki delujejo na enem samem računalniku.

V sodobnih operacijskih sistemih so procesi glede dostopa do podatkov med seboj čim bolj ločeni. V večini primerov je to zaželjeno, saj ne želimo, da bi en program lahko nezaželeno in nepričakovano vplival na delovanje drugih programov. To pomeni, da poljuben proces ne more enostavno brati ali spreminjati podatkov drugih procesov. Vendar je velikokrat zaželeno, da več različnih procesov med seboj deli podatke. Primer so razhroščevalniki, ki potrebujejo popoln dostop do pomnilnika nekega drugega procesa. V ta namen lahko operacijski sistemi nudijo razne mehanizme, ki procesom omogočajo deljenje podatkov.

2.1 Splošni mehanizmi medprocesne komunikacije

Danes mehanizme, ki omogočajo medprocesno komunikacijo, najdemo tako rekoč v vsakem širše uporabljenem operacijskem sistemu. Čeprav vsak operacijski sistem medprocesno komunikacijo arhitekturno izvaja na lasten način, vseeno obstajajo splošni principi o tem, kako lahko operacijski sistem omogoča dvema procesoma deljenje podatkov. V tem podpoglavju so na kratko predstavljeni pogosto uporabljeni principi, ki jih lahko najdemo v skoraj vsakem sodobnem operacijskem sistemu.

Komunikacija preko datotek - Prva ideja je kar komuniciranje preko datotek. V datoteko lahko piše in iz datoteke lahko bere poljuben proces, ki ima za to ustrezne pravice. Datoteke so podprte v vsakem običajnem operacijskem sistemu in večina programskih jezikov ponuja vgrajene in enostavne knjižnice s katerimi lahko pišemo in beremo iz datotek. Tak pristop ima veliko slabosti. Potrebna je sinhronizacija, saj ne želimo, da dva procesa hkrati pišeta v datoteko. Takšna oblika prenosa podatkov je počasna (saj pišemo in beremo najverjetneje s trdega diska).

Deljeni pomnilnik (Shared memory) - Podobna ideja, ki se znebi večino teh slabosti, je deljeni pomnilnik. Namesto branja in pisanja iz datoteke tu procesi berejo in pišejo v nek del pomnilnika, ki jim ga je dodelil operacijski sistem in je dostopen samo točno določenim procesom. S tem se hitrost pisanja in branja poveča, saj uporabljamo pomnilnik. Še vedno je potrebna sinhronizacija med procesi. Deljeni pomnilnik velja za najhitrejšo vrsto prenosa podatkov, saj je branje podatkov tujega procesa enako hitro kot branje lastnih podatkov. Sinhronizacija in uvedba dodatnih izboljšav, kot je na primer medpomnilnik, upočasni prenos podatkov, še posebej če v komunikaciji sodeluje veliko procesov.

Vtičnice (Sockets) - Danes zelo razširjena oblika medprocesne komunikacije so vtičnice (angl. sockets), ki so uporabljene na primer v interne-

tnem protokolu. Poznamo veliko število različnih vrst vtičnic, vsem pa je skupno, da omogočajo komunikacijo 1-proti-1. Vtičnice omogočajo neposredno komunikacijo med dvema procesoma in velikokrat ponujajo dodatne storitve, kot so zagotovilo o uspešnem prenosu (na primer pri TCP vtičnicah) in uvedba medpomnilnika za bolj učinkovito pošiljanje večjega števila sporočil. Za izmenjavo sporočil vtičnice uporabljajo medpomnilnik, pomnilnik določene velikosti, v katerem hranijo poslana sporočila, dokler jih prejemniki ne preberejo. Zaradi enostavne uporabe, ki ne potrebuje dodatne sinhronizacije, so vtičnice dober način prenosa podatkov, ko je potrebna hitra komunikacijo s čim manj dela.

Cevi (Pipes) - Uporabniki operacijskega sistema Linux, ki so vešči uporabe lupine, verjetno poznajo cevi (v ukazni lupini uporabljene z operatorjem |), ki preslikajo standardni izhod enega procesa na standardni izhod drugega. To omogoča na primer veriženje ukazov v Linuxovi ukazni lupini, vendar ni ustrezno za prenos velike količine podatkov, saj je pisanje na standardni vhod in branje s standardnega izhoda precej zamudno (še posebej, ko ne poznamo dolžine in oblike podatkov in je za to potrebno brati po en znak na enkrat). Nadgradnja tega sistema so poimenovane cevi, kjer sta vhod in izhod programa preslikana preko datoteke posebne vrste v operacijskem sistemu.

Ključavnice in semaforji (Locks, semaphores) - Med medprocesno komunikacijo lahko štejemo tudi razne sisteme, ki so pogosto uporabljeni v večnitnih programih. To so ključavnice in semaforji, ki ne omogočajo prenosa velike količine podatkov (pri ključavnicah gre le za en bit) in so uporabljene za sinhronizacijo procesov.

Splošni pregled mehanizmov medprocesne komunikacije s stališča enostavnosti in hitrosti je predstavljen na sliki 2.1.

Mehanizem	Hitrost	Enostavnost uporabe	Število procesov
Komunikacija preko datotek	Počasna	Enostavna	Poljubno mnogo
Deljeni pomnilnik	Zelo hitra	Zapletena	Poljubno mnogo
Vtičnice (Sockets)	Hitra	Enostavna	1-proti-1
Cevi (Pipes)	Počasna	Enostavna	1-proti-1

Slika 2.1: Pregled mehanizmov medprocesne komunikacije.

2.2 Serializacija

Ko pošiljamo podatke med različnimi procesi so ti na mediju, ki se uporablja za prenos, predstavljeni kot ničle in enice. Takšni podatki v večini primerov predstavljajo nekaj bolj konkretnega, na primer številko ali niz črk. Če želimo podatke prenašati med različnimi procesi, jih je najprej potrebno spremeniti v binarno obliko [21]. Na prvi pogled se to morda zdi preprosto. Vsi podatki so v pomnilniku predstavljeni kot zaporedje enic in ničel in tudi večina programskih jezikov omogoča preprosto pretvorbo v binarne tipe. Vendar ta rešitev deluje le, če procesa, ki izmenjujeta podatke, te podatke tolmačita na povsem enak način, na kar se ne moremo vedno zanašati. Različni programski jeziki ne bodo nujno razumeli dvojiških podatkov na enak način. Še več, tudi različni prevajalniki ali različne različice istega prevajalnika lahko dvojiške podatke razumejo na drugačne načine, saj nekateri standardi programskih jezikov (primer je standard za jezik C++ [26]) le zelo okvirno definirajo, kako naj bodo določeni tipi predstavljeni v pomnilniku. Pogosto se podatki razlikujejo v dolžini (v 32 bitnih arhitekturah so kazalci 32 bitni, v 64 bitnih 64 bitni) ali urejenosti bitov (najbolj pomemben bit prvi ali zadnji). Vendar tudi zagotovilo, da programi podatke razumejo na enak način ni dovolj. Nekaterih podatkovnih tipov namreč sploh ni mogoče smi-

selno pošiljati. Razredi, ki vsebujejo kazalce (na primer povezani seznami) ne bodo delovali, če jih enostavno prekopiramo v nek drug proces z drugačno pomnilniško strukturo. Potrebno jih je pretvoriti v drugačno obliko, na primer v enolično identifikacijsko številko objekta, na katerega se kazalec nanaša [28].

Rešitev tega problema je serializacija - enoten postopek, ki nek programski tip enolično preslika v njegovo binarno obliko. Poleg tega lahko napredne tehnike serializacije uvedejo tudi druge prednosti, kot so:

Vzvratna združljivost - Recimo, da se razvijalec programske opreme odloči, da trenutno uporabljenemu formatu sporočil v novi različici programa doda novo polje, vendar želi preko istega sistema še vedno pošiljati tudi staro različico sporočil (ki jih pošiljajo uporabniki stare verzije programa, ki še niso namestili posodobitve). V takem primeru mora pretvorba sporočil iz binarne v programsko obliko vedeti, za katero verzijo sporočila gre, da ga lahko pretvori v pravilni objekt.

Učinkovitost - Za hiter prenos podatkov je zaželeno, da so prenesena sporočila čim krajša. Če pošiljamo številke, ki so dolge največ 128 bitov, vendar so v večini sporočil veliko krajše, je bolj učinkovito sporočilo serializirati v formatu *dolžina števila + število*. Uporabimo lahko tudi kompresijske algoritme, če je čas stiskanja dovolj kratek. Mnoga sporočila vsebujejo opcijska polja, ki jih je možno v dobro zasnovani serializirani obliki izpustiti.

Samo opisnost - Za nekatere programe je bolj kot učinkovitost pomembno to, da jih razume katerikoli sistem, tudi tisti, ki ne poznajo postopka serializacije oziroma oblike sporočila. Primer je format JSON [15], ki je berljiv tudi za človeka. S tem je nastalo sporočilo večje, saj mora vključevati tudi podatke o tem, kakšne podatke sporočilo vsebuje. Sporočilo vsebuje podatke o načinu serializacije, da ga je možno smiselno pretvoriti.

Splošnost - Sistemi lahko omogočajo definiranje poljubnih podatkovnih tipov, ali neposredno v programskem jeziku ali z uporabo posebnih jezikov za opisovanje tipov (če gre za komunikacijo med programi, ki so napisani v različnih programskih jezikih).

Mnogi sodobni programski jeziki ponujajo vgrajene rešitve za serializacijo. S tem enostavno rešimo probleme pri prenosu kazalcev ter zagotovimo, da bodo sporočila delovala ne glede na različico programskega jezika ali prevajalnika. Vendar te načini serializacije velikokrat ne podpirajo prenosa podatkov med programi, napisanimi v različnih programskih jezikih. Java na primer podpira serializacijo vseh objektov, katerih razred vključuje vmesnik *java.io.Serializable* [24].

Danes je programerjem na voljo mnogo že izdelanih knjižnic za serializacijo, ki ponujajo napredne storitve. Zelo razširjene so knjižnice za serializacijo v datoteke oblike JSON [15] in XML [29]. Prav tako obstaja veliko knjižnic za pretvorbo v binarna sporočila. Primer je knjižnica Protocol Buffers, ki jo je razvilo podjetje Google [8].

2.3 Rešitve za medprocesno komunikacijo

Programerjem, predvsem tistim, ki so osredotočeni na visoko-nivojske koncepte, se danes ni več potrebno ukvarjati z nizko-nivojskimi mehanizmi medprocesne komunikacije in neposrednimi sistemskimi klici za njihovo uporabo. Na voljo imajo veliko število paketov in knjižnic, ki komunikacijo olajšajo in nudijo tudi mnoge dodatne storitve, ki niso na voljo direktno v operacijskem sistemu. Glavne izmed teh storitev so:

Vgrajena serializacija - Omogoča pošiljanje struktur, ki jih nudi programski jezik (na primer objektov v objektno usmerjenih programskih jezikih). Programerju ni potrebno skrbeti, kako bo te objekte pretvoril v prenosljivo binarno obliko. Pošilja lahko kar običajne objekte, ki so nato njemu nevidno serializirani. Pri nekaterih knjižnicah je potrebno

najprej definirati posebne objekte, ki jih bomo pošiljali, da jih knjižnice znajo serializirati.

Več-platformnost - Podprtih je lahko več različnih programskih jezikov in več različnih operacijskih sistemov. Poslani podatki so pretvorjeni v ustrezno obliko, ki jo razumejo vsi podprti operacijski sistemi in programski jeziki. Uporabljene so oblike komunikacije, ki so združljive z različnimi operacijskimi sistemi.

Arhitektura komunikacije - Procesi lahko med seboj komunicirajo na več različnih načinov. Najbolj preprost, ki ga neposredno podpira večina operacijskih sistemov, je komunikacija 1-proti-1 (neposredna komunikacija med dvema procesoma). Tak način prenosa podatkov ni vedno optimalen. Ena izmed priljubljenih oblik komunikacije je način pošiljateljev in naročnikov (angleško publish/subscribe). Pri takem načinu sporočanja se prejemniki sporočil naročajo ali na neke teme, ki opisujejo vrsto sporočil, ali na sporočila, ki ustrezajo določenim pogojem (na primer le sporočila ki vsebujejo številke ali le sporočila določene dolžine ali oblike). Pošiljatelji sporočil nato le naslovijo sporočilo na določeno temo (pri prvem načinu) ali podatke enostavno pošljejo (pri drugem načinu). Sistem nato poskrbi, da sporočilo prejmejo le tisti prejemniki, ki ga želijo prejeti. Pošiljatelju pri tem ni treba vedeti, koliko prejemnikov (če sploh kakšnega) ima poslano sporočilo.

Dodatna orodja - Ta na primer omogočajo pregled vseh programov, ki med seboj komunicirajo, trenutnega stanja sistema ali beleženje in pregled nad napakami in statistiko sistema.

2.4 Izbrane rešitve za medprocesno komunikacijo v operacijskem sistemu Linux

Diplomska naloga obsega tudi pregled in primerjavo nekaterih obstoječih rešitev tako z izdelanim sistemom kot druge z drugo. Spodaj so predstavljene tri izbrane rešitve, ki so primerjane v diplomski nalogi. Vse so splošno uporabne in se uporabljajo v praksi.

2.4.1 Lightweight Communication and Marshalling

Lightweight Communication and Marshalling (LCM) je sistem za medprocesno komunikacijo prvotno razvit za uporabo v robotskih sistemih, kjer je sporočanje med raznimi komponentami zelo pogosto [22]. Sprva je bil razvit leta 2006. Njegov razvoj se kot odprto-kodni projekt aktivno nadaljuje tudi danes. Pri izdelavi sistema LCM so avtorji dali velik poudarek na enostavnost uporabe in na združljivost s pogosto uporabljenimi platformami in programskimi jeziki. LCM tako poleg vseh treh najbolj pogosto uporabljenih operacijskih sistemov, torej Windows, Linux in OS X podpira še vsak operacijski sistem združljiv s standardom POSIX (portable operating system interface). Podprti programski jeziki so C, C++, C#, Java, Lua, MATLAB, Python in Vala [6].

Arhitekturno se LCM od ostalih primerjanih sistemov razlikuje po tem, da za prenašanje sporočil med različnimi procesi ne uporablja centralnega procesa, ki bi skrbel za naslavljanje sporočil. Sistem uporablja protokol UDP (user datagram protocol), ki je pogosto uporabljen pri pošiljanju sporočil skozi omrežja in svetovni splet. Sporočila so poslana vsem procesom ki uporabljajo sistem LCM, za zavračanje neželenih sporočil skrbijo procesi sami. Protokol UDP podpira takšen način pošiljanja sporočil (multicast način). Poleg tega je UDP podprt v vsakem pogosto uporabljenem operacijskem sistemu, saj je nujen za uporabo svetovnega spleta. Druga pomembna lastnost protokola UDP je, da UDP ne zagotavlja dostave sporočil. V praksi to pomeni, da protokol ob visoki nasičenosti komunikacijskih poti (ko procesi

pošiljajo veliko količino podatkov) nekatera sporočila zavrže. Ta lastnost je za nekatere sisteme ugodna, saj izboljša splošno zmogljivost sistema. Tisti sistemi, ki želijo zagotovilo o dostavi prav vseh sporočil morajo to zagotovilo implementirati sami, saj LCM tega ne podpira.

LCM za organizacijo pošiljanja sporočil uporablja sistem pošiljateljev in naročnikov. V poslanih paketih je zapisano, kateri temi pripadajo. Prejemniki sporočil prejmejo vsa sporočila, poslana znotraj sistema in sami zavržejo tista, ki jih ne zanimajo. Takšen pristop zmanjša vpliv števila prijavljenih procesov na neko temo. Sistemi, zasnovani na centralnem procesu namreč velikokrat vsakemu naročniku pošljejo lastno kopijo sporočila, torej čas pošiljanja sporočila vezanega na neko temo narašča linearno s številom naročnikov prijavljenih na to temo.

Razvijalci sistema LCM so razvili tudi lasten sistem za serializacijo. Sistem podpira ustvarjanje uporabniško definiranih tipov z uporabo definicijskih datotek, ki uporabljajo sintakso podobno programskemu jeziku C. Uporabljena serializacija zagotavlja kompatibilnost med podprtimi operacijskimi sistemi in programskimi jeziki.

LCM ponuja vgrajena orodja, s katerimi lahko razvijalci spremljajo potek pogovorov med procesi, kar je za komunikacijske sisteme dokaj redko. Vsa LCMjeva orodja z grafičnimi vmesniki so napisana v Javi, kar omogoča prenosnost med operacijskimi sistemi. Orodja omogočajo beleženje sporočil v dnevniških datotekah in enostaven pregled nad temi zapisi z možnostjo filtriranja po temah. Možen je tudi pregled nad sporočili v realnem času s podporo za uporabniško napisane vtičnike, ki omogočajo na primer vizualizacijo sporočil [22].

LCM je bil prvotno izdelan in uporabljen za namene ekipe na inštitutu MIT (Massachusetts Institute of Technology), ki je leta 2007 sodelovala na preizkušnji avtonomnih vozil DARPA (Defense Advanced Research Projects Agency) urban challenge [23, 22]. Danes ga uporabljajo tudi mnoge druge organizacije, ki so navedene na spletni strani projekta [6].

2.4.2 D-Bus

D-Bus je sistem za medprocesno komunikacijo, prvotno razvit kot enoten sistem medprocesne komunikacije za programe, ki delujejo na operacijskem sistemu Linux. D-Bus je zgolj specifikacija protokola z pripadajočo referenčno implementacijo, ki ni mišljena za splošno uporabo. Za protokol obstaja veliko število implementacij v raznih stanjih podpore in razvoja. Razvite implementacije podpirajo veliko število programskih jezikov [2]. Podprti so predvsem operacijski sistemi, ki podpirajo standard POSIX. Različica z podporo za operacijski sistem Windows že deluje, vendar je še v razvoju. [1]

Ključni princip protokola D-Bus je komunikacija preko vodila, kar omogoča enostavnejše pošiljanje in prejemanje sporočil, namenjenih več prejemnikom. Vodilo v visoko nivojskem smislu predstavlja nek medij, preko katerega lahko komunicira poljubno število procesov. Nizkonivojsko je implementiran kot proces, ki skrbi za pravilno dostavo sporočil. D-Bus uporablja dve ločeni vrsti vodil. Prvo je sistemsko vodilo, ki je dostopno vsem uporabnikom sistema in je namenjeno prenašanju sistemskih dogodkov. Drugi tip je uporabniško vodilo, ki je ločeno za vsakega uporabnika sistema.

D-Bus podpira veliko število dodatnih lastnosti. Implementira lastno serializacijo, pri čimer podpira različne vrste sporočil. Najbolj preprosta vrsta so signali, ki imajo enak namen kot signali v operacijskem sistemu, torej sporočanje, da se je nek dogodek zgodil. Glavni način prenosa podatkov je omogočen s podporo za klic oddaljenih funkcij (RPC). Pošiljatelj prejemniku pošlje podatke, ki služijo kot argumenti za prejemnikovo funkcijo, prejemnik na to odgovori z rezultatom funkcije. D-Bus da velik poudarek tudi na varnost. Če nek proces z omejenimi uporabniškimi pravicami kliče oddaljeno funkcijo, ki potrebuje večje pravice, ta ne bo izvedena. Mogoč je tudi nadzor dostopa do vodil in avtentikacija pošiljateljev in prejemnikov sporočil.

Velik problem D-Busa je hitrost pošiljanja in prejemanja podatkovnih sporočil [30, 13], predvsem zaradi velike režije, izvedene nad poslanimi podatki.

Trenutno poskušajo nekateri razvijalci operacijskega sistema Linux D-

Bus vključiti kar v jedro operacijskega sistema [17], kar naj bi po njihovem mnenju izboljšalo hitrost delovanja. Cilj razvijalcev je bila vključitev v jedro že leta 2014, vendar se to do aprila 2015 še ni zgodilo [18].

2.4.3 Robot Operating System

Kljub imenu Robot operating system (slovensko operacijski sistem za robote) ROS ni operacijski sistem v tradicionalnem smislu. Gre za zbirko orodij in programskih knjižnic ki ponujajo nekatere, vendar ne vse, funkcionalnosti ki jih običajno nudijo operacijski sistemi, med njimi tudi mehanizme za medprocesno komunikacijo. Poleg tega ROS ponuja še mnogo drugih funkcionalnosti, ki so potrebne za razvoj programske opreme, ki deluje na robotih. Sistem torej ni namenjen za splošno programsko opremo, temveč le za tisto, namenjeno uporabi v robotskih sistemi. Kljub temu je povsem mogoča uporaba knjižnic za medprocesno komunikacijo tudi v splošno namenskih programih [25].

Sistem omogoča uporabo sistema pošiljateljev in naročnikov. Poleg tega omogoča definicijo storitev: komunikacijskih modelov, kjer nek proces drugemu pošlje neke podatke in pričakuje od drugega procesa odgovor na podlagi teh podatkov (klic oddaljenih funkcij).

ROS ponuja razvijalcem veliko število pomožnih orodji, med njimi tudi orodje, ki lahko izriše graf vseh procesov, kjer povezave med procesi ponazarjajo, kateri procesi med seboj delijo podatke [25].

Sistem je nekoliko težje vključiti v že obstoječe projekte, saj proces dela s knjižnicami sistema večinoma sledi predpisanemu procesu, ki za prevajanje programov in upravljanje z odvisnostmi uporablja lastna orodja (oziroma spremenjene različice obstoječih orodij, na primer CMake). Prav tako ni velikega poudarka na kompatibilnosti z različnimi sistemi ali programskimi jeziki. Podpora je povsem zagotovljena le za operacijski sistem Linux, pri čimer različne različice sistema uradno podpirajo prav določeno izdajo določene distribucije (Ubuntu). Mac OS X, Android in ostale distribucije Linuxa so podprte le delno. Povsem podprti so programski jeziki C++ (le

do različice iz leta 2003), Python in Lisp. Eksperimentalno so podprti tudi programski jeziki C#, Go, Haskell, Java, Julia, Lua, Smalltalk, R in Ruby. Raven podpore in dokumentacije za te programske jezike je nižja kot za C++ in Python [20, 16].

ROS je zelo priljubljen in široko uporabljen sistem. Po podatkih avtorjev je podprt na več kot 45 različnih vrstah robotov [9] in ima več kot 3 milijone skupnih prenosov datotek za razvijalce [19].

Poglavje 3

Razviti sistem: Echo

V sklopu diplomske naloge je bil razvit enostaven sistem oziroma knjižnica za medprocesno komunikacijo, ki je bil poimenovan Echo. Sistem Echo deluje na operacijskem sistemu Linux in trenutno ponuja knjižnice za programski jezik C++. Sistem deluje na podlagi zaledne storitve operacijskega sistema (angl. daemon), ki skrbi, da so sporočila dostavljena procesom, katerim so namenjena, ter za ustvarjanje, nadziranje in zapiranje vtičnic operacijskega sistema Linux, na katerih je sistem zasnovan. Echo nudi le preprosto serializacijo, vendar omogoča enostavno integracijo z obstoječimi binarnimi serializacijskimi rešitvami, na primer FlatBuffers (nadgradnja Protocol Buffers, osredotočena na hitrost) [4]. Sistem podpira arhitekturo pošiljateljev in naročnikov: proces se lahko naroči na enega ali več kanalov in prejema sporočila poslana na te kanale. Prav tako pošiljatelji sporočil ne pošiljajo neposredno procesom, temveč le kanalom. V tem poglavju bo ta sistem predstavljen.

3.1 Motivacija

Danes je večina programskih storitev glede večprocesnosti napisana monolitno, torej tako, da tečejo le na enem procesu, morda z uporabo večnitnosti. Vendar je včasih koristno, če se program razdeli na več manjših procesov. S

tem se lahko izboljša zanesljivost sistema. Če en del programa odpove bo s tem odpovedal le proces, ki je odgovoren za ta del, medtem ko bodo ostali procesi lahko nadaljevali (seveda le, če del, ki je odpovedal, ni bil ključnega pomena za potek programa) ali vsaj varno zaključili izvajanje. Olajšana je tudi izdelava programa, saj je možno posamezne dele izdelati v različnih programskih jezikih ali z uporabo različnih programskih knjižnic.

Kljub prednostim, ki jih ponuja delitev v več manjših procesov, na trgu ni veliko knjižnic, ki omogočajo lahko sporazumevanje med različnimi procesi. Večina knjižnic, ki ponujajo storitve za medprocesno komunikacijo ne daje poudarka na hitrost in enostavnost, ali so izdelane posebej za uporabo v robotskih sistemih in ne v splošno namenskih programskih rešitvah. Cilj diplomske naloge je izdelava sistema, namenjenega predvsem za zmogljive interaktivne sisteme. Tu je potrebna visoka hitrost prenosa (za prenos slik in video posnetkov) ter hitra odzivnost na uporabnikove ukaze tudi takrat, ko je komunikacijski kanal že zasičen z podatki.

Razviti programski sistem naj bi bil primeren za splošno uporabo v vseh programskih rešitvah, ki potrebujejo enostavno medprocesno komunikacijo, ki je bolj osredotočena na enostavno uporabo in hitrost kot na veliko število dodatnih funkcionalnosti.

Eden izmed ciljev je tudi primerjava takšnega sistema z obstoječimi rešitvami, predvsem glede na sledeča kriterije.

Preprostost arhitekture - Sistem naj bo čimbolj preprost, s čim manj odvisnostmi na zunanje knjižnice. Prav tako naj bo preprost za razširitev, če bo potrebna dodatna funkcionalnost. Osnovna zgradba sistema naj bo čim bolj enostavna in razumljiva.

Hitrost - Sistem mora biti dovolj hiter, da prenos informacij ne vpliva na uporabniško izkušnjo. Ker je sistem razvit za prenos podatkov v interaktivnih sistemih je potreben kratek odzivni čas in velika pasovna širina prenosa podatkov.

Enostavna uporaba - Sistem naj bo čim bolj enostaven za uporabo s strani

razvijalcev aplikacij. To vključuje tako namestitev in vključitev knjižnice v poljuben programski projekt, kot tudi samo uporabo v programski kodi.

3.2 Uporabljene tehnologije

Sistem Echo je bil razvit v programskem jeziku C++. Dodatne knjižnice niso bile uporabljene, saj je eden izmed ciljev sistema enostavna namestitev, ki jo dodatne odvisnosti otežijo. Za mehanizem medprocesne komunikacije so bile uporabljene vtičnice, in sicer direktno z uporabo sistemskih klicev. Za spremljaje dogodkov na vtičnikih je bil uporabljen sistemski klic *epoll*, ki vrne podatke o dogodkih, ki so se zgodili na vtičnicah (sprejeta sporočila, napake pri povezavi ipd.).

Dodatne knjižnice za serializacijo niso vključene, vendar je bil sistem izdelan z namenom, da jih končni uporabniki lahko po želji vključijo. Preverjeni sta bili knjižnici FlatBuffers [4] in Cap'n Proto [27], ki tako kot sistem omogočata enostavno uporabo in hitro delovanje.

3.3 Opis sistema

Echo je zasnovan na konceptu pošiljateljev in naročnikov (angleško publish/subscribe), kjer se lahko procesi naročajo na kanale. Vsak kanal ima edinstveno ime. Procesi lahko komunicirajo le preko kanalov. Vsak proces se lahko prijavi na poljubno število kanalov. Vsakič, ko proces pošlje sporočilo na kanal, bo to sporočilo poslal vsem procesom, ki so na ta kanal naročeni. Za obdelavo sporočil vsak proces za vsak kanal, na katerega je naročen, definira povratno funkcijo, ki se bo klicala nad podatki sporočil, naslovljenih na ta kanal.

Ob prvem zagonu sistema obstaja le en kanal: nadzorni kanal. Ta kanal sprejema le sporočila posebne oblike, definirana v sistemu. Preko teh nadzornih sporočil lahko povezani procesi ustvarjajo nove kanale, se na kanale

prijavljajo in odjavljajo ter poizvedujejo po obstoječih kanalih.

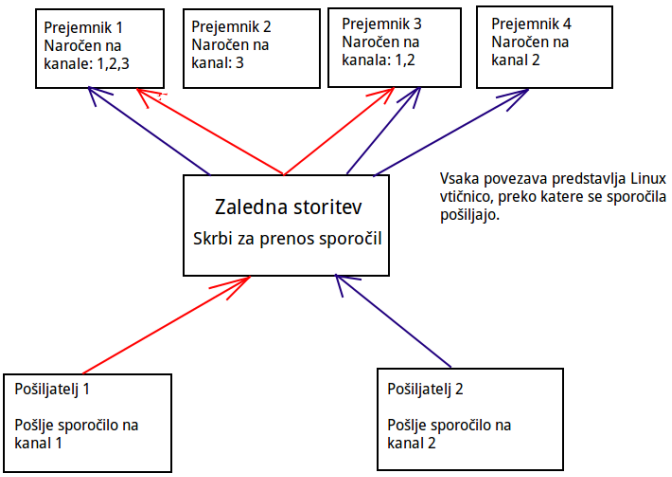
Za nadzor nad kanali in usmerjanje sporočil skrbi zaledna storitev, ki hrani seznam vseh povezanih procesov in kanalov, na katere so prijavljeni. Tako lahko posreduje sporočila le tistim procesom, katerim so namenjena. Zaledna storitev je tudi edina, ki bere sporočila na nadzornem kanalu in izvaja zahteve nadzornih sporočil (torej ustvarjanje kanalov in naročanje procesov). Echo je torej zasnovan centralno, z enim procesom, ki deluje v ozadju in skrbi za prenos sporočil. Takšna zasnova ima prednosti in slabosti. Ker za vso komunikacijo skrbi le en proces, postane ob napaki v tem procesu celoten sistem v trenutku neuporaben. Poleg tega procesi ne komunicirajo neposredno, ampak preko posrednika, kar podaljša čas prenosa sporočil. Po drugi strani je takšna arhitekturna rešitev preprosta, kar lahko zmanjša čas prenosa podatkov. Poleg tega je enostavna za razvoj in nadaljnje nadgradnje. Prav tako lahko z uporabo enega samega programa, ki ima vedno pogled nad celotnim procesom pošiljanj sporočil uvedemo dodatne lastnosti, kot je različna prioriteta. Ko centralni proces prejme sporočilo z visoko prioriteto (na primer uporabnikov vnos, na katerega želimo čim hitrejši odziv) lahko ustavi procesiranje manj pomembnih sporočil in se takoj posveti pomembnemu.

Na nivoju operacijskega sistema so za komunikacijo uporabljene Linux vtičnice (UNIX Domain Sockets) v asinhronem načinu pošiljanja in prejemanja, ki omogočajo učinkovito komunikacijo 1-proti-1 med procesi na istem sistemu.

Okvirna shema arhitekture sistema je prikazana na sliki 3.1.

3.3.1 Zaledna storitev

Zaledna storitev operacijskega sistema, oziroma daemon, predstavlja glavni del razvitega sistema. Zaledna storitev skrbi za operacije nad vtičnicami, povezovanje s procesi in za prenos sporočil od pošiljatelja do vseh zaželenih prejemnikov. Ob zagonu zaledna storitev ustvari nadzorno vtičnico. Preko te vtičnice le sprejema zahteve po priklopu procesov, ki želijo komunicirati preko storitve. Ko se nov proces želi povezati na to vtičnico s sistemskim klicem



Slika 3.1: Shema arhitekture sistema Echo.

connect(), storitev ustvari novo vtičnico, preko katerega bo od sedaj naprej komuniciral z novim procesom. Zaledna storitev torej za komuniciranje z n procesi potrebuje $n+1$ vtičnic: nadzorno in po eno za vsak proces.

Za spremljanje dogodkov na vtičnicah je uporabljen Linuxov sistem za obveščanje o dogodkih *epoll*, s katerim lahko spremljamo, katere povezane vtičnice vsebujejo neobdelane podatke. Zaledna storitev v glavni zanki najprej sprejme vse nove epoll dogodke in jih obdela glede na to, kateri vtičnici pripadajo in kakšne vrste so. Če gre za napake, se nanje poskuša primerno odzvati (izpis napake ali zaključek programa, če gre za kritično napako). Ob podatkovnih dogodkih ali ustvari nov vtičnik, če gre za dogodek na nadzornem vtičniku, ali obdela prejeto sporočilo, če gre za dogodek na vtičniku, ki pripada povezanemu procesu. Zaledna storitev najprej prebere vsa sporočila, ki so jih odjemalci storitvi poslali, in nanje ustvari odgovore. Nato odgovore pošlje. Hkratno pošiljanje vseh sporočil je bolj učinkovito, saj je možno z enim sistemskim klicem na vtičnico hkrati zapisati več sporočil. Izjema je le, če sporočila, ki jih želimo poslati v velikosti presežejo določeno mejo. V tem primeru se pošljejo takoj, saj neposlana sporočila zasedajo prostor v pomnil-

niku. Potek glavnih algoritmov zaledne storitve je predstavljen na slikah 3.2 in 3.3.

3.3.2 Odjemalci

Programi, ki uporabljajo izdelani sistem za komunikacijo, morajo za uspešno komuniciranje z zaledno storitvijo podpirati sledeče lastnosti:

Povezovanje s storitvijo - Klicati morajo ustrezne sistemske klice za ustvarjanje in povezovanje na vtičnice. Za povezovanje s storitvijo je potrebno poznati njeno enolično določeno ime, ki je izbrano ob zagonu storitve.

Tvorjenje sporočil - Iz programskih tipov kot so števila in nizi morajo biti zmožni tvoriti podatkovna sporočila, ki jih zaledna storitev razume. Tu gre ali za nadzorna sporočila, ki imajo točno določeno obliko, ali za podatkovna sporočila, ki so sestavljena iz kratke glave in binarnih podatkov.

Tvorjenje nadzornih sporočil - Z njimi lahko ustvarjajo kanale in se nanje prijavljajo in odjavljajo.

Pošiljanje sporočil na pravilen kanal - Nadzorna na nadzorni kanal, podatkovna na tisti, na katerega so sporočila naslovljena. Kanal, kateremu je sporočilo namenjeno je zapisan v glavi sporočila.

Sprejemanje sporočil in klicanje povratnih metod - Implementirati morajo metodo, s katero bodo ob prejemu sporočila klicane povratne funkcije. Povratne funkcije omogočajo enostavno obdelavo podatkov.

Vse te storitve morajo biti z uporabniškega vidika implementirane čim bolj preprosto. Cilj diplomske naloge je izdelava sistema, ki je čim bolj visokonivojski in enostaven za uporabo. Uporabniku torej detajli implementacije ne smejo biti vidni. V ta namen sta implementirana dva različna razreda.

```
Ustvari nadzorno vtičnico
Dodaj ustvarjeno vtičnico v seznam nadziranih vtičnic
dokler Storitev teče
    Preberi dogodke na vseh nadziranih vtičnicah
    za vsak prebran dogodek
        če je dogodek na nadzorni vtičnici potem
            če dogodek opisuje priključitev klienta potem
                Ustvari novo vtičnico in jo dodaj med nadzirane vtičnice.
                Po tej vtičnici se bodo izmenjevali podatki
                s priključenim procesom
            drugače če dogodek opisuje odključitev klienta potem
                Zapri klientovo vtičnico
                Odstrani vtičnico iz seznama nadziranih
            drugače če dogodek opisuje napako na nadzorni vtičnici potem
                Izpiši obvestilo o napaki
                Zaključi izvajanje
            konec če
        drugače če če je dogodek na katerikoli drugi vtičnici potem
            Obdelaj sporočila na vtičnici
            (Podrobneje predstavljeno na sliki 3.3)
        konec če
    konec za
    Pošlji vsa sporočila v vrsti za pošiljanje
konec dokler
```

Slika 3.2: Zaledna storitev: glavna zanka.

```
Iz vtičnice preberi glavo sporočila,
ki vsebuje dolžino podatkov in naslov (kanal) sporočila
Iz vtičnice preberi toliko podatkov, kot je dolžina sporočila
Iz podatkov in glave sestavi sporočilo
če je sporočilo namenjeno nadzornemu kanalu potem
    Izvedi akcijo, ki jo nadzorno sporočilo zahteva
    To je na primer prijava ali odjava na kanal
    če se odjemalec prijavlja na kanal potem
        dodaj odjemalca na seznam prijavljenih na ta kanal
    drugače če se odjemalec odjavlja od kanala potem
        Odstrani odjemalca iz seznama prijavljenih na ta kanal
    konec če
drugače če je sporočilo namenjeno drugim kanalom potem
    Iz glave preberi kanal
    Podatke vsebovane v sporočilu dodaj na vrsto za pošiljanje vsem
    klientom, ki so prijavljeni na ta kanal
    če je v vrsti za pošiljanje preveliko število podatkov potem
        Takoj pošlji vse podatke
    konec če
konec če
```

Slika 3.3: Zaledna storitev: obdelava posameznega sporočila.

Prvi je nizkonivojski in uporabniku skrit. Drugi je visokonivojski in služi kot ovojnica okoli nizkonivojskega. Tega neposredno uporablja uporabnik.

Uporabniku se v procesu komunikacije nikoli ni potrebno zavedati procesa pretvorbe iz osnovnih tipov v dvojiški zapis. Ta poteka v ozadju, medtem ko uporabnik ves čas uporablja le tipe na katere je navajen.

Povezovanje z zaledno storitvijo potrebuje le sistemski klic za povezovanje na storitveno vtičnico. Potrebno je vedeti ime storitvene vtičnice, ki je podano ob zagonu zaledne storitve.

Tvorjenje nadzornih sporočil uporabniku ni omogočeno neposredno. Ta sporočila imajo namreč točno določeno obliko in točno določen namen, na primer naročanje na kanale, ustvarjanje novih kanalov ali preklic naročnine. Za to je veliko bolj smiselno tvorjenje sporočil abstrahirati z uporabo namenskih funkcij. Primer je funkcija *subscribe()*, ki ustvari nadzorno sporočilo za prijavo na nek kanal in ga pošlje preko nadzornega kanala.

Kanali so uporabnikom predstavljeni kot nizi. Vendar so v ozadju, torej v zaledni storitvi, ki upravlja s kanali, predstavljeni kot številke. Za pretvorbo med nizom črk in številko odjemalec pošlje nadzorno sporočilo, ki vsebuje določen niz. Nato kot odgovor prejme identifikacijsko številko kanala, na katero lahko pošilja sporočila. To številko je potrebno le vključiti na ustrezno mesto v glavi sporočila, da zaledna storitev razume, komu je sporočilo namenjeno. Podobna pretvorba med zapisom, berljivim za ljudi in zapisom, primernim za računalnike je prisotna na primer tudi pri internetu, kjer poteka pretvorba med spletnimi naslovi in IP številkami.

Klic povratnih metod je implementiran s preprosto preslikavo iz številke kanala v seznam povratnih funkcij. Vsakič, ko odjemalec prejme sporočilo, ga le posreduje povratnim funkcijam. Vsak odjemalec lahko za vsak kanal definira več povratnih funkcij, če je to z vidika organizacije kode uporabniku potrebno.

3.4 Serializacija

Serializacija, ki jo uporablja sistem, je preprosta. Podpira le osnovne podatkovne tipe int, long, char, string in binarne podatke fiksne dolžine. Pretvorba v binarni zapis je dosežena s preprostim kopiranjem pomnilniške predstavitve podatkov. Takšna predstavitev ima mnogo pomanjkljivosti, vendar je v praksi primerna za uporabo na enem samem sistemu pri programih, napisanih v enem samem programskem jeziku. Pošiljanje naprednih struktur je podprto z uporabo zunanjih knjižnic, ki podatke serializirajo v dvojiško obliko.

Poleg binarnih podatkov vsako sporočilo vsebuje še preprosto glavo, v kateri je najprej zapisan znak za začetek sporočila, nato kanal sporočila in dolžina podatkov.

Takšna serializacija je zadostna za veliko število programske opreme. Tu gre predvsem za programe, ki delujejo lokalno na enem samem računalniškem sistemu in so napisani v skupnem programskem jeziku. To je tudi edini način, v katerem je knjižnica trenutno uporabna, torej napredna serializacija, razen za potrebe dodatne funkcionalnosti kot je vzvratna in vnaprejšnja združljivost ali za podporo tipov s kazalci, ni nujno potrebna.

Če uporabniki potrebujejo bolj napredno serializacijo, sistem podpira katerokoli serializacijsko knjižnico, ki rezultat vrne v binarni obliki. Testirana je bila knjižnica FlatBuffers [4], Googlova nizko-latenčna visoko-hitrostna serializacijska knjižnica, ki je nastala kot nadgradnja široko uporabljene knjižnice Protocol Buffers[8].

Shema formata serializacije je predstavljena na sliki 3.4.

3.5 Uporaba

Uporabniki za komunikacijo uporabljajo razred *CommunicationChannel*. Ta razred omogoča enostavno sporočanje preko določenega kanala. Uporabnik ustvari instanco razreda, ki ji poda tip, ki ga želi prenašati, ime razreda in povratno funkcijo. Konstruktor razreda nato pošlje potrebna nadzorna sporočila za poizvedbo po kanalu in prijavo nanj. Nato uporabnik v glavni

Slika 3.4: Shema formata serializacije knjižnice Echo.

zanki svojega programa kliče funkcijo *wait()*, ki skrbi za pošiljanje in prejemanje sporočil ter za klice povratnih funkcij. Za pošiljanje sporočil uporabi funkcijo *send()*, ki najprej pretvori objekt, ki ga uporabnik želi poslati v sporočilo, in to sporočilo doda na seznam sporočil, ki bodo poslana ob naslednjem klicu *wait()*. Pošiljanje sporočil je torej asinhrono.

Razred *CommunicationChannel* je šablonski (angl. template) razred z enim argumentom (ki predstavlja tip sporočil, ki se preko kanala pošiljajo), kar omogoča definicije različnih implementacij glede na to, kakšen tip podatkov želi uporabnik pošiljati. Podprtih je večina osnovnih tipov ter dva posebna primera. Če uporabnik uporablja lastno knjižnico ali funkcijo za serializacijo, kot šablonski argument poda kazalec tipa void in število, ki predstavlja dolžino podatkov. Če uporabnik kot argument poda razred, ki ni podprt s strani knjižnice, mora ta razred vsebovati metodo za serializacijo in metodo, ki vrne dolžino serializiranega sporočila. To omogoča pisanje lastnih serializacijskih metod za uporabnikove razrede

Uporabnik lahko za komunikacijo tudi direktno uporablja nizkonivojski razred *client()*, vendar to ni potrebno, saj zgoraj omenjeni razred nudi vse,

kar običajen uporabnik potrebuje.

Za povratne funkcije je uporabljen C++11 razred *std::function*. S tem je omogočena tudi uporaba anonimnih funkcij. Razred *std::function* je bil v jezik C++ dodan s standardom C++11, torej je za uporabo knjižnice potrebna uporaba prevajalnika, ki ta standard podpira. Uporaba sistema je prikazana na slikah 3.6 in 3.5. Shematski prikaz poteka uporabe je na voljo na sliki 3.7.

```
//Primer uporabe
//Posiljatelj
#include <stdio>
//include knjiznice izdelanega sistema
#include "communication_channel.h"

int main(int argc, char** argv){
//Ustvarimo povezavo z zaledno storitvijo.
//Argument pove ima vticnika storitve,
// na katero se povezujemo.
auto client = make_shared<echolib::Client>("test.sock");

//Prijavimo se na kanal channel1
//navesti moramo tip sporocila,
//in zgoraj ustvarjeno povezavo.
echolib::CommunicationChannel<std::String> channel(client,
                "channel1");

//Sporocilo dodamo v vrsto za posiljanje.
channel.send("Pozdravljen!");

//Posljemo vsa sporocila v vrsti.
//S tem locimo dodajanje podatkov,
//ki jih zelimo poslati in
//dejansko izvajanje posiljanja.
//argument pove, koliko milisekund naj
//klic blokira, ko caka na sporocila
//ker le posiljamo, je na 0.
channel.wait(0);
```

Slika 3.5: Prikaz enostavne uporabe sistema za pošiljanje sporočil.

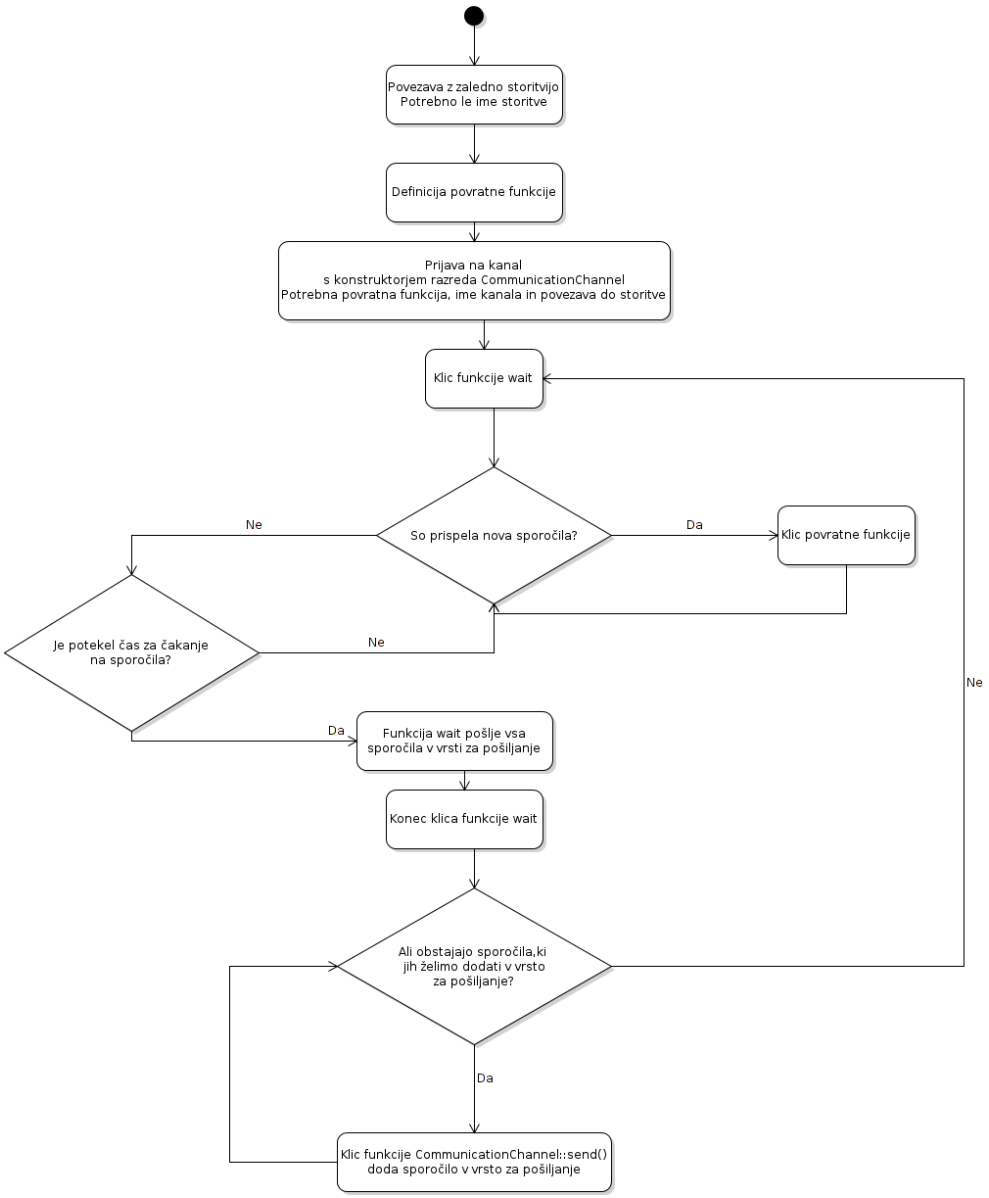
```
//primer prejemnika
#include <stdio>
//include knjiznice izdelanega sistem
#include "communication_channel.h"

int main(int argc, char** argv){
//Ustvarimo povezavo z zaledno storitvijo.
//Argument pove ima vticnika storitve,
// na katero se povezujemo.
auto client = make_shared<echolib::Client>("test.sock");
//Povratna funkcija
//ki se bo klicala ob
//vsakem prejetem sporocilu
auto callback = [](String sporocilo){
        std::cout << "Prejel sem sporocilo: " << sporocilo;
};

//Prijavimo se na kanal channel1
//navesti moramo tip sporocila,
//zgoraj ustvarjeno povezavo in povratno funkcijo.
echolib::CommunicationChannel<int> channel(client,
                "channel1",
                callback);

//Cakamo na sporocila
//zanka se bo izvajala,
//dokler je s povezavo vse v redu
//argument pove, koliko milisekund naj
//klic blokira, ko caka na sporocila
while(channel.isConnected())
        channel.wait(0);
}
```

Slika 3.6: Prikaz enostavne uporabe sistema za prejemanje sporočil.



Slika 3.7: Shematski prikaz splošne uporabe sistema Echo.

Poglavje 4

Meritve zmogljivosti

Izdelani sistem je bil primerjan s tremi izbranimi sistemi, ki so danes pogosto uporabljeni pri izdelavi programskih rešitev, ki uporabljajo medprocesno komunikacijo. Primerjava je bila izvedena glede na sledeče lastnosti, ki so pomembne pri izdelavi interaktivnih sistemov.

Režija - Sistem za obdelavo vsakega sporočila potrebuje nekaj časa. To pomeni, da je pošiljanje veliko majhnih sporočil lahko veliko počasnejše kot pošiljanje istega števila podatkov v manjšem število velikih paketov

Pasovna širina - Koliko podatkov lahko na sekundo največ prenaša nek sistem.

Zakasnitev - Koliko časa potrebuje en paket, da pride od pošiljatelja do prejemnika. To je zelo pomembno v aplikacijah, ki z uporabnikom komunicirajo v realnem času. Uporabnik ne želi velikega zamika med njegovimi ukazi in začetkom izvajanja teh ukazov na zaslonu. Če pošiljamo velike količine podatkov je zakasnitev lahko zelo odvisna od pasovne širine. Zakasnitev je pomembna pri prenosu podatkov med procesi, ki tečejo na različnih računalnikih. Pri lokalni komunikaciji je zelo majhna, za človeško zaznavanje celo neopazna.

Cena dodatnih prejemnikov - Kako na hitrost vpliva število procesov, ki se med seboj sporazumevajo. Nekateri sistemi namreč pošljejo le eno

sporočilo, ki ga lahko bere vsak proces. Drugi vsakemu procesu pošljejo lasten izvod besedila.

Izguba paketov - Nekateri sistemi pri prenosu velikega števila podatkov nekatere zavržejo, če jih ne morejo obdelati dovolj hitro.

Enostavnost - Zaželjeno je, da je sistem enostaven, tako za namestitev kot za uporabo v programu. Zaželjena je tudi dobra dokumentacija.

Splošnost - Sistem naj bo možno uporabiti v čim več programskih jezikih in na čim več operacijskih sistemih.

Združljivost - Sistem naj bo čim bolj združljiv z drugimi orodji, na primer s knjižnicami za serializacijo, ali z naprednimi lastnostmi programski jezikov.

Dodatna funkcionalnost - Poleg pošiljanja sporočil mnogi sistemi nudijo tudi dodatne storitve. Pogosto podpirajo klice oddaljenih podprogramov ali dodatna orodja, s katerimi lahko spremljamo promet ali razne karakteristike.

S strani uporabnika sta najbolj opazni lastnosti zakasnitev in pasovna širina. Zakasnitev je posebej pomembna pri programih, ki se na uporabnikove vnose odzivajo v realnem času, medtem ko je pasovna širina pomembna pri programih ki pošiljajo velike količine podatkov. Sistem je namenjen za okolje, kjer sta zahtevani tako nizka zakasnitev kot velika pasovna širina. Torej sta ti meritvi pri ovrednotenju sistemov najbolj pomembni. Drugi cilj izdelanega sistema je enostavna uporaba s strani programerja, torej je tudi to pomemben vidik pri ovrednotenju izbranih sistemov.

Poleg tega uporabljeni sistem ni edini faktor, ki vpliva na hitrost komunikacije. Večina sistemov ali direktno v sistemu ali v implementaciji komunikacije s strani operacijskega sistema uporablja medpomnilnike, ki hranijo sporočila preden so obdelana. Velikosti teh medpomnilnikov lahko vplivajo na hitrost in druge zmogljivosti, predvsem na izgubo paketov pri sistemih, ki

to dovoljujejo. Te nastavitve so še posebej pomembne pri sistemu LCM, kjer velikost medpomnilnika neposredno vpliva na število izgubljenih paketov.

Takšne meritve niso povsem reprezentativne za ovrednotenje dejanskih zmogljivosti sistemov. Testno okolje je namreč podvrženo operacijskemu sistemu, predvsem v smislu menjavanja procesov ter dodeljevanja pomnilniškega časa. Ker so vsi sistemi podvrženi istim pogojem so kljub temu primerne za uporabo pri primerjavi sistemov.

4.1 Merjeni sistemi

Poleg izdelanega sistema Echo so bili glede na zgornje lastnosti ovrednoteni še sistemi D-Bus, ROS in sistem LCM. Od teh treh sistemov je LCM tisti, ki je v zasnovi najbližji sistemu Echo, saj sta oba osredotočena na enostavnost tako v arhitekturi kot pri uporabi. Sistema ROS in D-Bus sta veliko bolj kompleksna, in ponujata množico storitev, ki niso primerljive z izdelanim sistemom. Primerjava s tema sistemoma je osredotočena zgolj na prenašanje enostavnih sporočil med več procesi.

Glede na opise in zgradbo sistemov je bilo pričakovano, da se bo Echo obnesel primerljivo s sistemom LCM. Sistem D-Bus je glede na mnenja in meritve mnogih uporabnikov počasen [13, 30], zato je bilo pričakovano, da se bo v meritvah hitrosti obnesel slabo, predvsem ko gre za velike količine podatkov. Sistem ROS je sicer zelo razširjen v uporabi, vendar ponuja veliko število dodatnih orodji in pri izdelavi ni imel hitrosti kot najpomembnejše zahteve [3], zato je bilo pričakovano, da se bo naš sistem obnesel hitreje.

Vsi sistemi, s katerimi je primerjan Echo imajo vgrajeno serializacijo. Za to je bilo pričakovano, da bo režija sporočil pri izdelanem sistemu manjša. Po drugi strani sporočila v sistemu Echo potujejo od pošiljatelja do zaledne storitve in šele potem do prejemnika. Pričakovano je bilo, da bo to negativno vplivalo tako na čas režije kot na čas pošiljanja.

4.2 Metodologija

Pomožni programi, ki izvajajo meritve so napisani v programskem jeziku C++, s sledečo osnovno zgradbo.

Uporabljena sta dva testna programa: en, ki pošilja sporočila in en, ki jih sprejema. To omogoča spremljanje hitrosti sistema glede na spreminjanje števila prejemnikov in pošiljateljev, ki sistem uporabljajo za komunikacijo. Nekateri sistemi namreč pošljejo sporočilo le enkrat za vse prejemnike, tako da tudi veliko število prejemnikov ne bi smelo znatno vplivati na hitrost prenosa. Drugi sistemi vsakemu prejemniku dostavijo lastno kopijo, kar poveča čas pošiljanja.

Povratna funkcija, ki se kliče ob sprejemu sporočila je namerno zelo preprosta: le šteje število sprejetih sporočil in izpiše čas in pasovno širino, ko prejme vse. Merimo namreč le komunikacijski sistem, zato želimo, da prejemniki kar se da hitro sprejmejo vsa sporočila. Pri testu za pasovno širino začnemo čas šteti po prvem prejetem sporočilu, s čimer zanemarimo zakasnitev in čas zagona sprejemnikov in pošiljateljev.

Preko vhodnih argumentov je možno nastavljati tako velikost sporočila kot število poslanih paketov. Praviloma je namreč prenos nekega števila podatkov hitrejši, čim manj paketov imamo, saj vsak paket doda nek čas procesiranja s strani sistema.

Če je velikost poslanih sporočil nastavljena na najmanjše možno, torej le na eno dvaintrideset-bitno število, je možno sistem uporabiti tudi za merjenje režije poslanih paketov. Za bolj dosledne meritve pri tem merimo celoten čas za obdelavo velike količine sporočil.

Sistem, uporabljen za zagon testnih primerov, ima sledeče lastnosti :

Procesor - Intel Pentium CPU B980 @ 2.40GHz x 2 .

Pomnilnik - 4,0 GB, od tega 3,6 GB prostega in 0,4 GB rezervirano za operacijski sistem.

Operacijski sistem - Ubuntu 14.04.3.

4.3 Rezultati

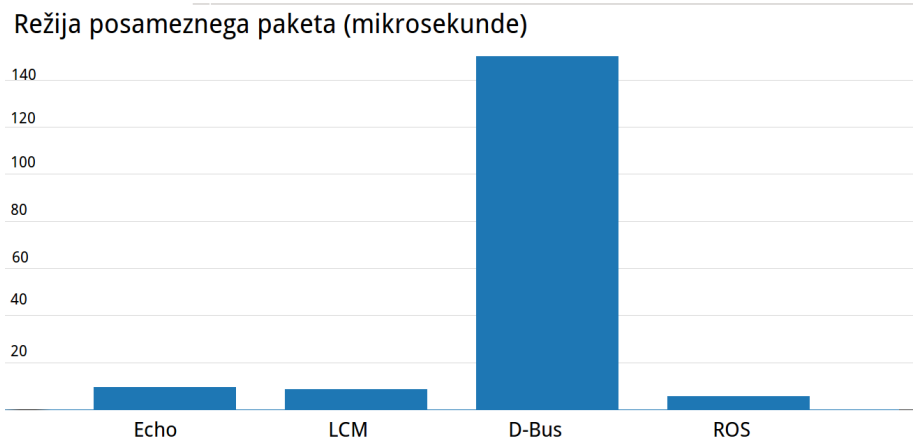
4.3.1 Režija

Režija posameznega paketa predstavlja čas, ki ga sistem za medprocesno komunikacijo porabi, da paket sprejme, pregleda in pošlje pravilnemu prejemniku. Čim bolj zapleten je komunikacijski sistem, tem večja bo režija. Visoka režija znatno vpliva na hitrost prenosa podatkov, kadar pošiljamo veliko število majhnih sporočil.

Pri ovrednotenju režije so se bili sistemi ROS, LCM in Echo zelo hitri, medtem ko je bil sistem D-Bus izredno neučinkovit. Izmerjeni rezultati so prikazani v tabeli 4.1 in grafu 4.2.

Echo	LCM	D-Bus	ROS
9,5 μs	8,6 μs	150 μs	5,6 μs

Slika 4.1: Izmerjene vrednosti režije.



Slika 4.2: Primerjava režije sistemov.

Najbolje se je kljub velikemu številu dodatne funkcionalnosti in vgrajeni serializaciji obnesel ROS. Verjetno je njegova serializacija pri enostavnih paketih (kot je poslani paket z le enim številom) izredno enostavna in

učinkovita. Drugi najboljši je sistem LCM, ki za dostavo paketov želenim prejemnikom ne uporablja posebnih algoritmov, ampak le način pošiljanja podatkov, ki je vgrajen v protokol UDP. Sistem Echo je tretji. Kljub enostavni zasnovi sistema je sporočilo potrebno pošiljati dvakrat, enkrat od pošiljatelja do zaledne storitve, nato od storitve do prejemnika, kar podvoji čas pošiljanja. Kljub temu je režija še vedno zelo majhna. Razlike med sistemi ROS, Echo in LCM so zelo majhne, le nekaj μs, kar je pri pošiljanju večjih sporočil skoraj neopazno.

Od ostalih treh sistemov izredno odstopa sistem D-Bus. To je bilo glede na prebrano literaturo [30, 13] pričakovano. D-Bus nad poslanimi sporočili izvaja veliko število dodatnih operaciji, tako pri serializaciji kot pri prejemanju in pošiljanju sporočil. Glede režije je kar okoli 15-krat počasnejši kot ostali primerjani sistemi, kar pomeni, da gotovo ni primeren za pošiljanje velikega števila majhnih sporočil.

4.3.2 Pasovna širina

Pasovna širina predstavlja največje število podatkov, ki jih je možno prenesti v nekem časovnem obdobju (običajno je predstavljena kot število megabajtov, ki jih je možno prenesti v eni sekundi, z enoto MB/s). V mnogih sistemih, ki uporabljajo medprocesno komunikacijo, je potrebna pasovna širina velikosti okoli 10MB/s [22]. Pasovna širina je posebno pomembna pri sistemih, ki prenašajo velike količine podatkov, na primer videoposnetke visoke ločljivosti. Mejo 10MB/s so dosegli vsi preverjeni sistemi, tako da so za običajno uporabo v sistemih, ki ne prenašajo velikega števila podatkov, lahko uporabljeni vsi preverjeni sistemi.

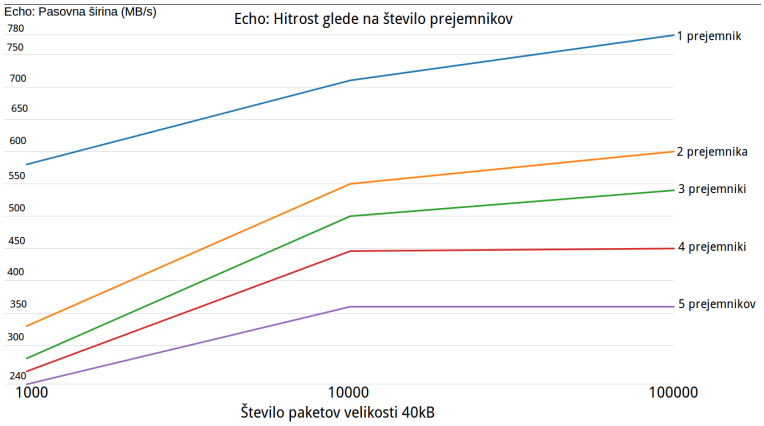
Na pasovno širino lahko vpliva tako število paketov, ki jih želimo prenesti, kot njihova velikost. Če prenašamo ogromno število majhnih paketov bo režija, potrebna za vsak paket, znatno vplivala tudi na hitrost prenosa podatkov. Sistemi lahko iz medija, ki se uporablja za prenos podatkov (na primer vtičnice) v enem sistemskem klicu preberejo več paketov na enkrat. Zato je hkratno pošiljanje večjega števila paketov lahko bolj učinkovito kot

pošiljanje majhnega števila. Pasovna širina je torej odvisna tako od velikosti posameznega paketa, kot od števila paketov, ki jih pošiljamo, zato so bile meritve izvedene pri različnih velikostih in pri različnem številu paketov. Na hitrost prenosa vpliva tudi velikost medpomnilnika mehanizma prenosa. Večji kot je medpomnilnik, hitrejši je prenos podatkov, vendar je velikost medpomnilnika omejena z velikostjo pomnilnika na testnem sistemu.

Prav tako lahko na pasovno širino prenosa vpliva število prejemnikov. Sistemi, zasnovani na centralnem procesu, ki skrbi za prenos sporočil, kot je tudi Echo, ponavadi vsakemu prejemniku dostavijo lastno kopijo sporočila. Nekateri sistemi, kot je LCM, pošljejo le eno kopijo sporočila, ki jo prebere vsak prejemnik, s čimer zmanjšajo vpliv dodatnih prejemnikov na hitrost sporočila. Kljub temu vsak dodaten prejemnik uporablja nekaj procesorskega časa, kar ima vpliv na hitrost prenosa sporočil. Pri rezultatih je izmerjena pasovna širina pri več prejemnikih predstavljena kot pasovna širina posameznega prejemnika, ne kot skupna pasovna širina.

Izdelani sistem Echo se je glede pasovne širine izkazal za izredno neenakomernega. Pri optimalnih pogojih (ki so se izkazali kot pošiljanje paketov velikosti 40kB) se je izkazal za izredno učinkovitega. Pri pošiljanju velikih (4MB) paketov, je dosegel hitrosti okoli 120MB/s, kar je nekoliko počasneje kot LCM, vendar hitreje kot ostala preverjena sistema. Na hitrost prenosa je po pričakovanju vplivalo število prejemnikov, vendar padec hitrosti ni bil linearen s številom dodatnih prejemnikov. To se je zgodilo zaradi dodeljevanja pomnilnika s strani operacijskega sistema in zaradi učinkovite uporabe hitrega medpomnilnika ob večkratnem kopiranju sporočil. Hitrost prenosa glede na število prejemnikov je prikazana na sliki 4.3. Hitrost prenosa pri enem prejemniku glede na velikost paketov je prikazana na sliki 4.4.

Ostali sistemi so se obnesli precej različno. ROS, ki se je pri režiji obnesel odlično, je bil pri pasovni širini slabši. Dosegel je hitrosti od 40MB/s do 50MB/s. Tudi pri pošiljanju majhnih sporočil velikih 400B je bil počasnejši od sistema Echo (ROS je dosegel 26MB/s, Echo 38MB/s). Pri pošiljanju bolj kompleksnih sporočil, ki vsebujejo podatke v oblike tabel, je režija pri



Slika 4.3: Primerjava pasovne širine sistema Echo glede na število prejemnikov.

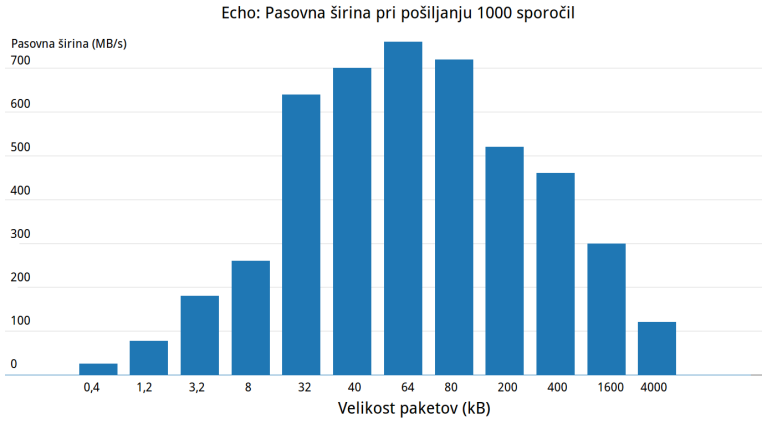
sistemu ROS torej večja.

LCM se je obnesel dobro, s hitrostmi do 300MB/s. Kljub nastavitvi velikega medpomnilnika za prenos sporočil (200MB) je bila opazna izguba paketov, predvsem ko se je sporočila pošiljajo večjemu številu prejemnikov. LCM se je obnesel najbolje v prenosu velikih paketov, s hitrostjo okoli 150 MB/s. Kljub uporabi UDP multicast načina za prenos sporočil je večje število prejemnikov vplivalo na hitrost prenosa, saj je vsak prejemnik za sprejem sporočila potreboval nekaj procesorskega časa.

Sistem D-Bus se je obnesel najslabše. Zaradi velike režije je pri prenosu majhnih sporočil skoraj neuporaben. Pri večjih sporočilih se je obnesel nekoliko bolje, s hitrostmi okoli 35MB/s, kar je najpočasneje od vseh primerjanih sistemov.

Zanimivo je, da je optimalna velikost sporočila pri vseh sistemih zelo podobna, in sicer okoli 64kB. Okoli te velikosti so vsi izmerjeni sistemi dosegli največjo možno pasovno širino.

Primerjava posameznih sistemov je prikazana na slikah 4.5 in 4.6. Meritve posameznih sistemov so predstavljene v tabelah 4.7, 4.8 in 4.9.

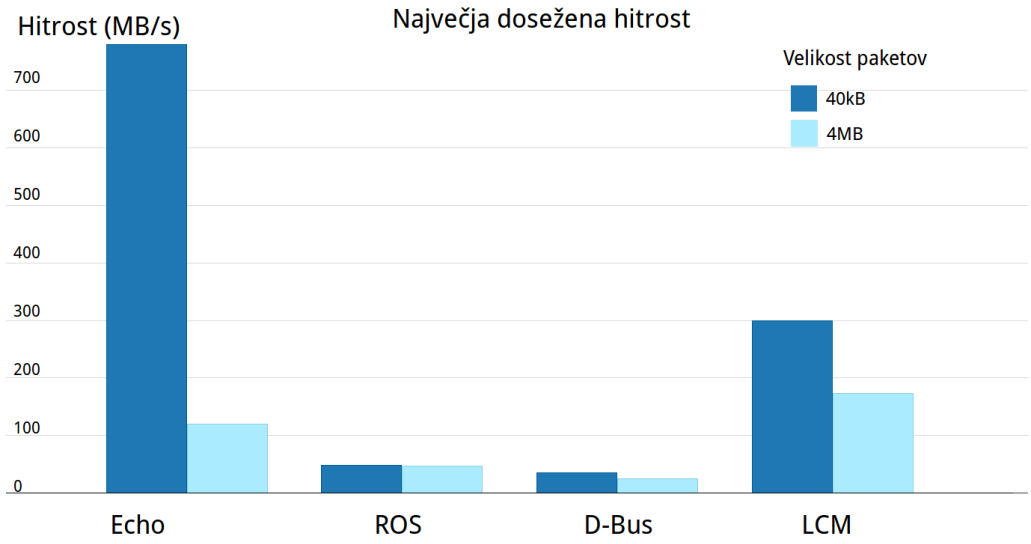


Slika 4.4: Primerjava pasovne širine sistema Echo glede na velikost paketov pri 1 prejemniku (1000 paketov).

4.3.3 Uporaba in namestitev

Ovrednoteni sistemi so, kar se tiče zahtevnosti pri uporabi in namestitvi, precej različni. Najbolj enostavna za uporabo sta LCM in Echo, ki ponujata tudi najmanj lastnosti in imata enostavnost kot enega izmed ciljev pri izdelavi. LCM sicer zahteva dodatne nastavitve omrežja, sicer je izguba paketov prevelika [7]. ROS je nekoliko bolj kompleksen za uporabo, saj ponuja več funkcionalnosti in zahteva uporabo lastnih orodij za upravljanje s projekti in izgradnjo programske kode. ROS tudi zahteva namestitev velikega števila knjižnic. Sistem ROS sicer ponuja dokumentacijo [11], ki olajša uporabo sistema. Najtežji za uporabo je D-Bus, saj je razdrobljen med mnogo različnih različic [2], od katerih vsaka zahteva svoj način namestitve in dela. Prav tako je sama uporaba v programski kodi precej bolj zahtevna, saj zahteva definicijo storitev v jeziku XML za vsako vrsto sporočil, ki jo želimo pošiljati. Poleg tega uradna implementacija nikoli ni bila mišljena za uporabo, saj je uradno D-Bus zgolj opis procesa prenosa podatkov, ki ga različno implementirajo različne različice [1].

ROS, D-Bus in LCM vsi ponujajo lastno rešitev za serializacijo. LCM in ROS tudi podpirata definicijo lastnih tipov. Echo namesto lastne knjižnice



Slika 4.5: Primerjava pasovne širine ovrednotenih sistemov.

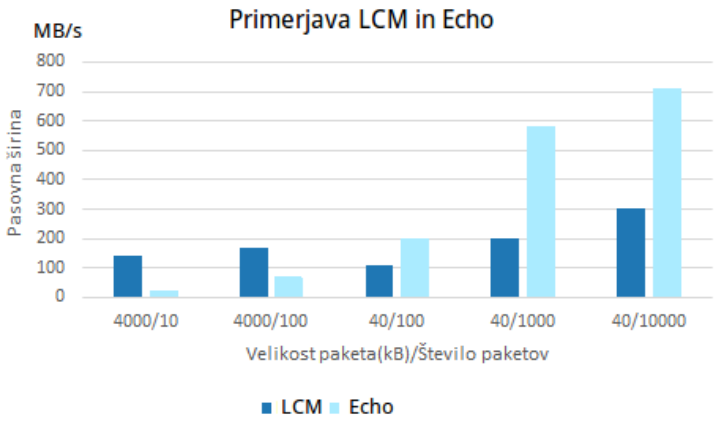
ponuja enostavno integracijo z obstoječimi knjižnicami ki ciljajo na hitrost, kot sta FlatBuffers [4] in Cap'n Proto [27].

LCM in D-Bus podpirata največ programskih jezikov in operacijskih sistemov. Oba podpirata tako Linux kot Windows ter veliko število programskih jezikov [2, 6].

Kratek pregled nekaterih lastnosti preverjenih sistemov je na voljo v tabeli 4.10.

4.4 Ugotovitve

Pri primerjavi so se dobro obnesli sistemi ROS, LCM in Echo, medtem ko je bil D-Bus znatno prepočasen tako s stališča latence kot s stališča prenosa podatkov. LCM ponuja konsistentno dobre hitrosti okoli 100 MB/s, vendar ponuja le prenos z možnostjo izgube paketov [22], ki je lahko v nekaterih programih nezaželena. ROS je počasnejši s hitrostmi okoli 50MB/s, vendar ponuja veliko dodatne funkcionalnosti in orodij. Izdelani sistem Echo je pri



Slika 4.6: Primerjava pasovne širine sistema LCM in izdelanega sistema.

optimalnih pogojih izjemno učinkovit, s pasovno širino okoli 700MB/s ob velikosti paketa 40kB. Ob paketih velikih okoli 4MB je nekoliko počasnejši kot LCM, vendar še vedno hiter, s pasovno širino okoli 120MB/s.

ROS, LCM in Echo so vsi enostavni za uporabo. Sistem ROS je sicer nekoliko bolj kompleksen za nastavitev in za postavitev razvojnega okolja, vendar ponuja odlično dokumentacijo. Sistem D-Bus je težji za uporabo.

Če je potrebna uporaba tako v operacijskem sistemu Linux kot v operacijskem sistemu Windows sta od naštetih sistemov primerna le LCM in D-Bus, čeprav poudarek razvoja knjižnice D-Bus ni na podpori operacijskega sistema Windows.

Sistemi ROS, D-Bus in LCM podpirajo tudi prenos podatkov preko omrežja. LCM za prenos podatkov uporablja omrežni protokol UDP, za to ga je možno uporabljati na običajnih omrežjih brez dodatnih nastavitev (razen nastavitev požarnih zidov ipd.). ROS za uporabo na omrežju potrebuje dodatne nastavitve omrežja [10]. D-Bus sicer ni bil mišljen kot protokol, ki bi podpiral prenos podatkov preko omrežja [5], vendar nekatere implementacije tak način prenosa podatkov podpirajo.

Največ dodatnih storitev podpira sistem ROS, ki ponuja tako prenos sporočil kot klice oddaljenih funkcij in opcijsko dovoljevanje izgube pake-

			Pasovna širina na prejemnika glede na število prejemnikov (MB/s)				
Velikost paketa	število paketov	Skupna velikost	1	2	3	4	5
400B	1000000	400MB	38	25	19	16	13
40kB	1000	40MB	580	330	280	260	240
40kB	10000	400MB	710	550	500	446	260
40kB	100000	4GB	780	600	540	450	360

Slika 4.7: Pasovna širina: Echo.

tov. Poleg tega ponuja tudi pomožne programe, ki razvijalcem pomagajo pri razvoju sistema. D-Bus tudi nudi dodatno funkcionalnost, saj nudi klic oddaljenih funkcij ter podporo za signale - čisto enostavna sporočila. LCM ne nudi dodatne funkcionalnosti pri pošiljanju sporočil, vendar nudi dodatna orodja za pomoč pri razvoju. Razviti sistem ne ponuja dodatnih funkcionalnosti, razen združljivosti z zunanjimi serializacijskimi knjižnicami.

Izdelani sistem Echo se je pri zastavljenih ciljih obnesel dobro. Dosegel je večinoma dobre, v nekaterih pogojih celo izvrstne rezultate v pasovni širini. Poleg tega je kljub uporabi centralnega procesa za izmenjavo sporočil dosegel nizek čas režije paketov, kar omogoča hiter odzivni čas. Doseženi so bili vsi cilji, zastavljeni pred izdelavo sistema. Sistem Echo je tako zelo uporaben v interaktivnih sistemih, saj dosega vse potrebne pogoje.

Velikost paketa	Število paketov	Skupna velikost	Pasovna širina (MB/s)
40kB	10	400kB	110
40kB	100	4MB	110
40kB	1000	40MB	200
40kB	10000	400MB	300
4MB	10	40MB	140
4MB	100	400MB	170
4MB	1000	4GB	173

Slika 4.8: Pasovna širina: LCM.

Velikost paketa	Število paketov	Skupna velikost	Pasovna širina (MB/s)
400B	1000000	400MB	26
40kB	10000	400MB	48
4MB	100	400MB	47

Slika 4.9: Pasovna širina: ROS.

Lastnosti/Sistemi	Echo	LCM	D-Bus	ROS
Namestitev	Enostavna	Enostavna	Zahtevna	Zahtevna
Težavnost uporabe v programu	Enostavna	Enostavna	Zahtevna	Enostavna
Podprti programski jeziki	Le C++	Veliko	Veliko	Veliko
Podprti operacijski sistemi	Linux	POSIX + Windows	Linux, delno Windows	Linux
Serializacija	Z uporabo knjižnic	Lastna	Lastna	Lastna
Potrebne dodatne knjižnice	Ne	Malo	Veliko	Veliko
Klic oddaljenih funkcij	Ne	Ne	Da	Da
Vključena orodja za pomoč pri razvoju	Ne	Da	Ne	Da

Slika 4.10: Primerjava dodatnih lastnosti sistemov.

Poglavje 5

Sklepne ugotovitve

V sklopu diplomskega dela je bil razvit preprost sistem za medprocesno komunikacijo, ki je dosegel cilje enostavnosti, tako v arhitekturi in programskem delovanju kot v uporabnosti. Kljub enostavni in centralizirani strukturi sistem deluje učinkovito, predvsem, ko je v izmenjavi sporočil udeleženih manjše število procesov. Prav hitra in enostavna namestitev ter splošno namenska uporabnost sistem ločita od velikega števila trenutno dostopnih rešitev (od preverjenih predvsem D-Bus in ROS). Ključni pomanjkljivosti sistema sta omejitev le na eno platformo (Linux) in le na en programski jezik (C++). Kljub temu sistem ustreza prvotnim specifikacijam, saj je namenjen za okolje, ki uporablja pretežno operacijski sistem Linux.

Sistem se je v testih zmogljivosti obnesel dobro. Po hitrosti je v določenih pogojih celo premagal ostale splošno dostopne rešitve, vendar ob ceni manjšega števila ponujenih lastnosti. Predvsem je imel prednost pri preverjanju režije (overhead) posameznih sporočil, saj uporablja enostavno in hitro serializacijo, brez dodatnega obdelovanja sporočil. Največja težava je nedosledna hitrost prenašanja sporočil glede na njihovo velikost.

Trenutno je sistem izdelan do take mere, da bi bil lahko uporabljen v katerikoli programski opremi, ki deluje na operacijskem sistemu Linux in deliti ali prejema podatke od drugih procesov, ki delujejo na istem sistemu. Za preizkus funkcionalnosti je bil poleg programov za meritev hitrosti izdelan

tudi preprost program za izmenjavo tekstovnih sporočil.

5.1 Nadaljnji razvoj

Zgoraj omenjeni ključni pomanjkljivosti sistema sta omejitev na le eno platformo (Linux) in na le en programski jezik (C++). Slednjo pomanjkljivost bi se dalo odpraviti, saj za mnogo programskih jezikov obstajajo mehanizmi za klice C++ funkcij. Primer je orodje SWIG (Simplified Wrapper and Interface Generator) [14, 12], ki ga je možno uporabiti za povezovanje C in C++ kode z velikim številom drugih programskih jezikov. Omejitev na operacijski sistem Linux je težje rešljiva. Sistemski klici za ustvarjanje in pošiljanje podatkov po vtičnicah so sicer podprti tudi v drugih POSIX podprtih operacijskih sistemih, vendar uporabljajo nekatere dodatke, ki so na voljo le v operacijskem sistemu Linux. Kompatibilnost z operacijskim sistemom Windows bi zahtevala nove funkcije, ki uporabljajo sistemske klice operacijskega sistema Windows. Problem je tudi uporaba sistemskega klica epoll za nadziranje dogajanja na vtičnikih, ki je povsem na voljo le v operacijskem sistemu Linux.

Problem slabše hitrosti pri velikih sporočilih bi lahko rešili z uvedbo algoritma, ki bi velika sporočila razdelil na majhne kose in vsak kos posebej poslal zaledni storitvi. To bi izboljšalo nizke hitrosti pošiljanja velikih sporočil.

Nova funkcionalnost bi bila lahko dodana predvsem z implementacijo prioritet sporočil. S tem bi lahko neko sporočilo zaledni storitvi povedalo, da je zelo pomembno in da ga je potrebno procesirati čim prej. S tem bi lahko ohranili odzivnost sistema na uporabnikove ukaze tudi ob pošiljanju velikega števila podatkov.

Izdelani sistem ima dokončano vso osnovno funkcionalnost, ki omogoča enostavno in učinkovito komunikacijo med več procesi. Enostavna zasnova sistema omogoča enostavno razširljivost z dodatno funkcionalnostjo, ki bi hitrost prenosa podatkov lahko še izboljšala. Izdelani sistem Echo je že povsem primeren za praktično uporabo.

Literatura

[1] (2015) DBus. Dostopno na: http://www.freedesktop.org/wiki/Software/dbus/ [Citirano 26.8.2015].

[2] (2015) DBus Bindings. Dostopno na: http://www.freedesktop.org/wiki/Software/DBusBindings/ [Citirano 26.8.2015].

[3] (2015) FAQ - ROS Wiki. Dostopno na: http://wiki.ros.org/FAQ#How_does_ROS_compare_in_terms_of_performance.3F [Citirano 28.8.2015].

[4] (2015) Flat Buffers: Main Page. Dostopno na: https://google.github.io/flatbuffers/index.html [Citirano 26.8.2015].

[5] (2015) Introduction To DBus. Dostopno na: http://www.freedesktop.org/wiki/IntroductionToDBus/ [Citirano 26.8.2015].

[6] (2015) LCM: Lightweight communications and marshalling(LCM). Dostopno na: http://lcm-proj.github.io/ [Citirano 26.8.2015].

[7] (2015) LCM: UDP multicast setup. Dostopno na: http://lcm-proj.github.io/multicast_setup.html [Citirano 28.8.2015].

[8] (2015) Protocol Buffers — Google Developers. Dostopno na: https://developers.google.com/protocol-buffers/ [Citirano 26.8.2015].

[9] (2015) Robots - ROS Wiki. Dostopno na: http://wiki.ros.org/Robots [Citirano 26.8.2015].

[10] (2015) ROS/NetworkSetup - ROS Wiki. Dostopno na: http://wiki.ros.org/ROS/NetworkSetup [Citirano 26.8.2015].

[11] (2015) ROS/Tutorials - ROS Wiki. Dostopno na: http://wiki.ros.org/ROS/Tutorials [Citirano 26.8.2015].

[12] (2015) Simple Wrapper and Interface Generator. Dostopno na: http://www.swig.org/ [Citirano 28.8.2015].

[13] H. Abbes in J.-C. Dubacq, "Analysis of peer-to-peer protocols performance for establishing a decentralized desktop grid middleware," v *Euro-Par 2008 Workshops-Parallel Processing*. Springer, 2009, str. 235–246.

[14] Beazley, Dave and SWIG Team and others, "Simplified wrapper and interface generator," 1995.

[15] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," 2014.

[16] K. Conley *et al.* (2015) Client Libraries - ROS Wiki. Dostopno na: http://wiki.ros.org/Client_Libraries [Citirano 26.8.2015].

[17] J. Corbet. (2014) The unveiling of kdbus. Dostopno na: https://lwn.net/Articles/580194/ [Citirano 26.8.2015].

[18] ——. (2015) The kdbuswreck. Dostopno na: http://lwn.net/Articles/641275/ [Citirano 26.8.2015].

[19] T. Foote, "ROS Community Metrics Report," 2014. Dostopno na: http://download.ros.org/downloads/metrics/metrics-report-2014-07.pdf

[20] T. Foote in K. Conley. (2010) REP 3 – Target Platforms (ROS.org). Dostopno na: http://www.ros.org/reps/rep-0003.html [Citirano 26.8.2015].

[21] J. S. Gray, *Interprocess communications in Linux*. Prentice Hall Professional, 2003.

[22] A. S. Huang, E. Olson, in D. C. Moore, "LCM: Lightweight communications and marshalling," v *Intelligent robots and systems (IROS), 2010 IEEE/RSJ international conference on.* IEEE, 2010, str. 4057–4062.

[23] J. Leonard, J. How, S. Teller, M. Berger, S. Campbell, G. Fiore, L. Fletcher, E. Frazzoli, A. Huang, S. Karaman *et al.*, "A perception-driven autonomous urban vehicle," *Journal of Field Robotics*, zv. 25, št. 10, str. 727–774, 2008.

[24] L. Opyrchal in A. Prakash, "Efficient object serialization in Java," v *Electronic Commerce and Web-based Applications/Middleware, 1999. Proceedings. 19th IEEE International Conference on Distributed Computing Systems Workshops on.* IEEE, 1999, str. 96–101.

[25] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, in A. Y. Ng, "ROS: an open-source Robot Operating System," v *ICRA workshop on open source software*, zv. 3, št. 3.2, 2009, str. 5.

[26] R. Smith *et al.*, "Working Draft, Standard for Programming Language C++," ISO, Tehnično poročilo N4296, November 2014. Dostopno na: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf

[27] K. Varda. (2015) Cap'n Proto: Introduction. Dostopno na: https://capnproto.org/ [Citirano 26.8.2015].

[28] S. J. White in D. J. DeWitt, "A performance study of alternative object faulting and pointer swizzling strategies," v *Proc. 18th Int. Conf. Very Large Data Bases, Vancouver, BC, Canada*, 1992.

[29] F. Yergeau, T. Bray, J. Paoli, C. M. Sperberg-McQueen, in E. Maler, "Extensible markup language (XML) 1.0," *W3C Recommendation*, zv. 4, 2004.

[30] B. Zores, "Embedded linux optimization techniques: How not to be slow," 2011, Embedded Linux Conference Europe 2011. Dostopno na: http://elinux.org/images/d/de/ELCE_2011_-_BZ_-_Embedded_Linux_Optimization_Techniques_-_How_Not_to_Be_Slow.pdf